**Analisi del Malware – Relazione Homework 3**

Studente: Andrea Pepe Matricola: 0315903

# Obiettivo

Analizzare il programma eseguibile “hw3.exe” utilizzando principalmente *Ghidra* e *OllyDbg* per determinare il codice di sblocco che rende funzionale il programma.

# Analisi di base

Il file eseguibile oggetto dell’analisi è in formato Portable Executable (PE), dunque un eseguibile per sistemi operativi Windows. Come prima operazione di analisi statica di base, è stata effettuata una ricerca delle stringhe utilizzate dal programma, in particolare usando il comando *strings* su sistema operativo Linux. Si riporta di seguito il risultato ottenuto, eliminando le stringhe non significative:

!This program cannot be run in DOS mode.

.text

P`data

P`.data

.rdata

0@.bss

.idata

.CRT

.tls

.rsrc

D$rTimef

D$v bef

D$zore

D$~Wind

ows

shut

down

D$@days

D$Ehour

D$Kminu

D$Otes

D$SUnlo

D$Wck c

D$[ode:

Escape from Windows v5 - M. Cesati, 2010, 2021

WARNING: there will be no shutdown without the proper unlock code!

0 seconds

%2ld seconds

Stop

Sorry, try again!

Shutdown time has come!

However, the unlock code is wrong.

If you want the full version of this wonderful tool,

you can get the unlock code for just ten bucks!

Ask to the nearest teacher around you!

EDIT

CreateWindow failed

BUTTON

SeShutdownPrivilege

InternalError

%s=%lu/0x%lx

DEBUG %s:%d

InternalError

Unknown error

\_matherr(): %s in %s(%g, %g) (retval=%g)

Argument domain error (DOMAIN)

Argument singularity (SIGN)

Overflow range error (OVERFLOW)

The result is too small to be represented (UNDERFLOW)

Total loss of significance (TLOSS)

Partial loss of significance (PLOSS)

Mingw-w64 runtime failure:

Address %p has no image-section

VirtualQuery failed for %d bytes at address %p

VirtualProtect failed with code 0x%x

Unknown pseudo relocation protocol version %d.

Unknown pseudo relocation bit size %d.

GCC: (tdm64-1) 9.2.0

AdjustTokenPrivileges

LookupPrivilegeValueA

OpenProcessToken

TextOutA

CloseHandle

CreateFileA

CreateFileMappingA

DeleteCriticalSection

EnterCriticalSection

ExitProcess

FatalAppExitA

GetCurrentProcess

GetCurrentProcessId

GetCurrentThreadId

GetFileInformationByHandle

GetLastError

GetModuleFileNameA

GetProcAddress

GetStartupInfoA

GetSystemTimeAsFileTime

GetTickCount

InitializeCriticalSection

IsDebuggerPresent

LeaveCriticalSection

LoadLibraryA

MapViewOfFile

QueryPerformanceCounter

SetUnhandledExceptionFilter

Sleep

TerminateProcess

TlsGetValue

UnhandledExceptionFilter

VirtualProtect

VirtualQuery

\_\_getmainargs

\_\_initenv

\_\_lconv\_init

\_\_p\_\_acmdln

\_\_p\_\_fmode

\_\_set\_app\_type

\_\_setusermatherr

\_amsg\_exit

\_cexit

\_initterm

\_iob

\_onexit

\_vsnprintf

abort

calloc

exit

fprintf

free

fwrite

malloc

memcpy

signal

strlen

strncmp

vfprintf

BeginPaint

CreateWindowExA

DefWindowProcA

DispatchMessageA

DrawTextA

EndPaint

ExitWindowsEx

GetClientRect

GetDlgItemInt

GetDlgItemTextA

GetMessageA

GetWindowLongA

KillTimer

LoadCursorA

LoadIconA

MessageBoxA

MoveWindow

PostQuitMessage

RedrawWindow

RegisterClassExA

SendDlgItemMessageA

SetDlgItemInt

SetDlgItemTextA

SetTimer

SetWindowLongA

ShowWindow

TranslateMessage

ADVAPI32.dll

GDI32.dll

KERNEL32.dll

msvcrt.dll

USER32.dll

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">

<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">

<security>

<requestedPrivileges>

<requestedExecutionLevel level="asInvoker"/>

</requestedPrivileges>

</security>

</trustInfo>

<compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">

<application>

<!--The ID below indicates application support for Windows Vista -->

<supportedOS Id="{e2011457-1546-43c5-a5fe-008deee3d3f0}"/>

<!--The ID below indicates application support for Windows 7 -->

<supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>

<!--The ID below indicates application support for Windows 8 -->

<supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}"/>

<!--The ID below indicates application support for Windows 8.1 -->

<supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/>

<!--The ID below indicates application support for Windows 10 -->

<supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}"/>

</application>

</compatibility>

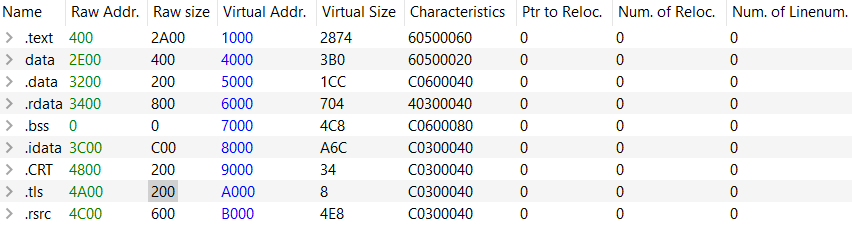
</assembly>

Dalla stringa “GDI32.dll” e dal nome di alcune API come, ad esempio, “GetMessageA” e “TranslateMessage” si desume che il programma abbia una GUI.

Inoltre, sono molto interessanti i nomi di alcune API che potrebbero essere state usate per ostacolare il reversing del codice del programma, adottando misure anti-debugging; in particolare: *GetTickCount, IsDebuggerPresent, LoadLibraryA, GetProcAddress, QueryPerformanceCounter, SetUnhandledExceptionFilter, UnhandledExceptionFilter.*

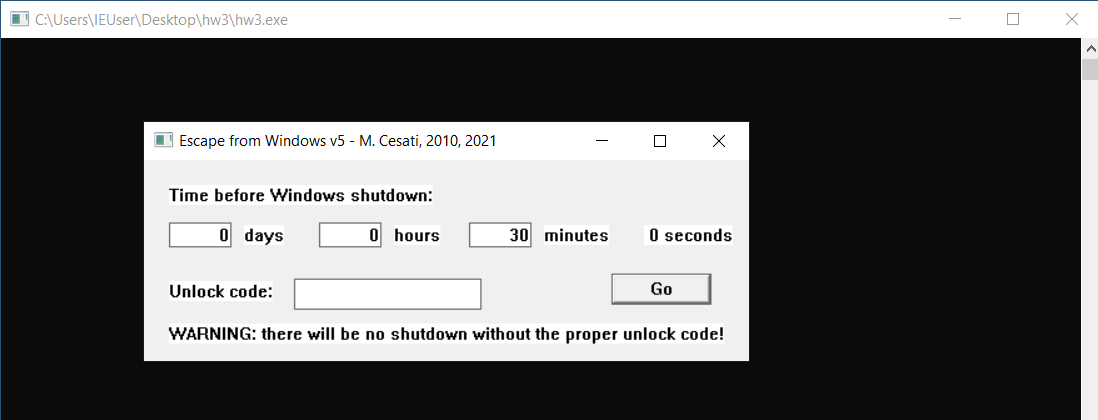
Infine, viene stampato tra le stringhe anche il contenuto di un file manifest in XML che descrive la compatibilità dell’eseguibile con le diverse versioni dei sistemi operativi Windows: sono supportati Windows Vista, 7, 8, 8.1 e 10. Infatti, usando il software *Resource Hacker*, tale file manifest risulta essere presente nella sezione dedicata alle risorse del file eseguibile.

Si è passati ad analizzare le testate del file eseguibile tramite l’utilizzo di *PE-Bear*: dalle informazioni riguardanti i SECTION\_HEADERS si è confrontata la raw size e la virtual size di ogni sezione. Sostanzialmente, i valori erano quasi sempre simili, fatta eccezione per la sezione .tls: infatti, come visibile dallo stesso *PE-Bear*, sono presenti nella TLS directory i riferimenti a 2 tls callbacks. Esse potrebbero essere usate per eseguire del codice prima che il debugger riesca a fermarsi sull’entry point del programma.



Pe-Bear: analisi dei Section Hdrs

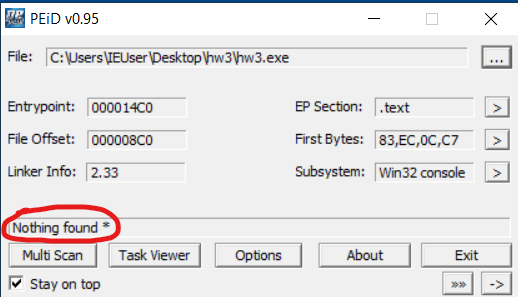
Si è proceduto con l’analisi dinamica di base e come prima cosa è stato lanciato l’eseguibile su una macchina virtuale con sistema operativo Windows 10 per osservarne il comportamento: viene aperta una finestra per programmare lo shutdown di Windows; non inserendo alcun codice di sblocco, allo scadere del countdown, appare una message box con un messaggio d’errore e il programma termina senza effettuare lo spegnimento del sistema.



Esecuzione del programma

Analizzando il processo in esecuzione tramite *ProcessExplorer* e *ProcessMonitor,* si nota che vengono effettuate diverse operazioni sul registro di sistema, ma non sembrano significative. Utilizzando *RegShot*, per fare un confronto tra lo status del registro di sistema prima di lanciare l’eseguibile e quello dopo averlo lanciato, se ne ha la conferma.

Un’analisi tramite *PEiD*, conferma che l’eseguibile non è stato compresso o offuscato tramite l’uso di un packer, così come era desumibile anche dalle informazioni ottenute usando *PE-Bear*:



Analisi con PEiD

Provando a lanciare il programma all’interno di *OllyDbg*, si nota che la finestra per programmare lo spegnimento di Windows non viene mostrata e, anziché rimanere running eseguendo il message loop, il programma termina. È evidente che sono presenti delle misure anti-debugging. Ci si pone quindi come obiettivo quello di individuare tali misure tramite l’uso dell’analisi statica avanzata con *Ghidra* combinata con *OllyDbg*, in maniera tale da inibirle e poter così effettuare l’analisi dinamica avanzata con il debugger.

# Analisi avanzata

Cercando nella sezione degli import l’API TranslateMessage() e guardandone i riferimenti, è stato individuato il message loop e quini la funzione *WinMain:*



Decompilato della funzione WinMain() in cui figurano già informazioni individuate in un secondo momento

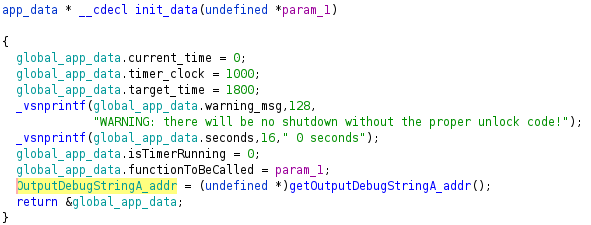
Ciò ha permesso di individuare facilmente la Window Procedure, ridenominata *WndProc.* Tuttavia, prima di inizializzare la struttura dati WNDCLASSEXA, viene invocata una funzione, la quale è stata ridenominata *getFileInfo*. In questa funzione, tramite l’API *GetModuleFilenameA,* si ottiene il path del file eseguibile del processo corrente. Se tale path esiste, si ottiene un handle aperto al file tramite l’invocazione a *CreateFileA,* che viene usato come parametro nella chiamata a *GetFileInformationByHandle*. In questo modo si ottengono in una struttura dati informazioni sul file e, in particolare, viene salvata in una variabile globale ridenominata *nSizeFileLow* la dimensione in bytes del file. A questo punto, viene invocata l’API *CreateFileMappingA* per creare ed ottenere un handle ad un file mapping object per il file eseguibile. Infine, tramite la *MapViewOfFile*, viene mappato la vista del file mapping object nell’address space del processo chiamante e il valore di ritorno, che corrisponde all’indirizzo iniziale della vista mappata, viene salvato in una variabile globale, ridenominata *startingAddress.*

È riportato di seguito il decompilato della funzione appena descritta:



Decompilato della funzione getFileInfo()

In seguito, nella *WinMain,* viene invocata una funzione (*init\_data*) che svolge il compito di inizializzare la struttura dati fondamentale contenente i dati dell’applicazione. Dunque, analizzandola, si individuano alcuni campi di tale struttura dati. Tuttavia, oltre a fare ciò, invoca un’altra funzione, la quale usa *LoadLibraryA* per caricare la dll kernel32.dll e *GetProcAddress* per ottenere l’indirizzo dell’API *OutputDebugStringA.* Tale indirizzo viene salvato in una variabile globale.



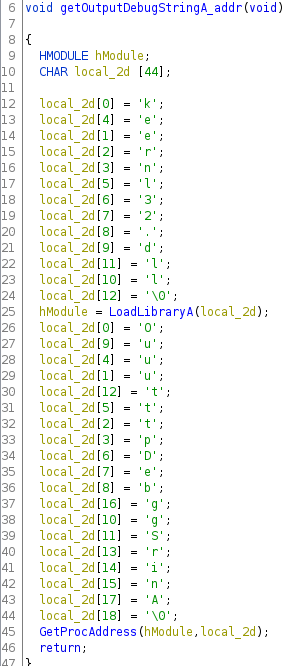
Decompilato della funzione init\_data()

L’analisi della funzione *getOutputDebugStringA\_addr()* è stata complicata dall’utilizzo di misure anti-disassembling; in particolare, è stata utilizzata la tecnica del disassemblaggio impossibile con il seguente schema:

|  |  |  |  |
| --- | --- | --- | --- |
| **EB** | **FF** | **C0** | **48** |

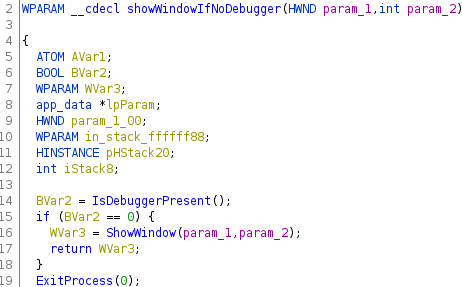
* EB FF 🡪 jmp -1; la prossima istruzione partirà dal byte FF;
* FF C0 🡪 inc eax; incrementa l’accumulatore di 1;
* 48 🡪 dec eax; decrementa l’accumulatore di 1.

Dopo aver opportunamente istruito Ghidra per facilitare il disassemblaggio della funzione, il decompilato è il seguente:



Decompliato della funzione getOutputDebugStringA\_addr()

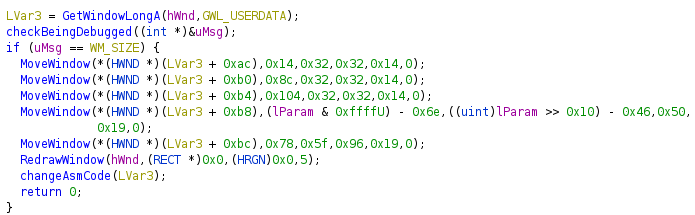
Procedendo nell’analisi della *WinMain*, si nota che, dopo aver creato la finestra, viene invocata la funzione ridenominata *showWindowIfNoDebugger:*



Decompilato della funzione showWindowIfNoDebugger()

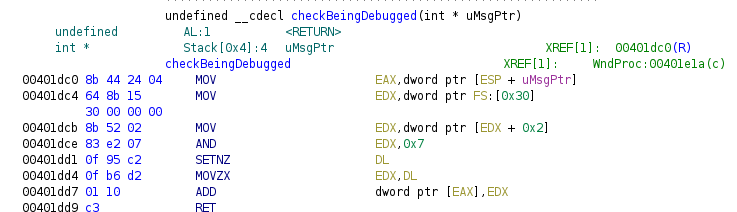
Tale funzione, invocando l’API *IsDebuggerPresent*, controlla se l’eseguibile è lanciato sotto il controllo di un debugger: se sì, termina il programma invocando *ExitProcess(0)*, altrimenti effettua la *ShowWindow* della finestra precedentemente creata. Risulta dunque ovvio che, per poter effettuare un’analisi tramite debugger, bisogna di certo inibire la chiamata ad *IsDebuggerPresent* oppure modificare il risultato che restituisce.

Prima di effettuare una patch dell’eseguibile, si decide di proseguire ulteriormente con l’analisi statica alla ricerca di eventuali ulteriori misure anti-debugger. Quindi, si passa ad analizzare la *WndProc*, di cui di seguito si riporta parte del decompilato.



Operazioni iniziali della WndProc e gestione di WM\_SIZE

Prima di discernere che tipo di messaggio è stato ricevuto dalla finestra, viene invocata una funzione, ridenominata in seguito *checkBeingDebugged*, che prende come parametro in input l’indirizzo di *uMsg*, ovvero il parametro della *WndProc* che identifica proprio il tipo di messaggio. Si riporta di seguito il disassemblato di tale funzione:



Disassemblato di checkBeingDebugged()

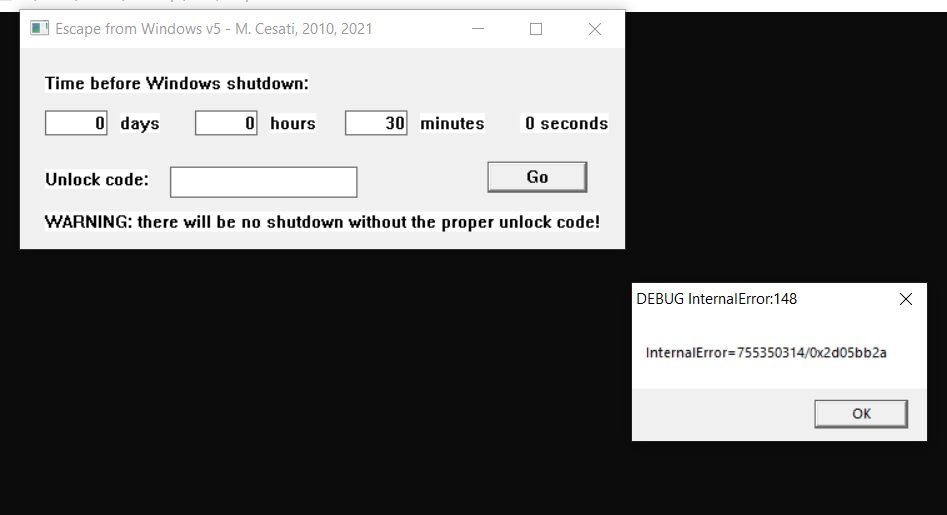
Con FS: [0x30], si accede alla struttura dati PEB (Process Environment Block); dopodiché, si prende il valore del campo presente ad offset 2 di tale struttura, ovvero il valore del campo *BeingDebugged*. Tale campo è pari a 1 se il processo è eseguito sotto il controllo di un debugger, 0 altrimenti. Quindi, nel caso in cui il processo sia debuggato, la funzione *checkBeingDebugged* modifica il valore di *uMsg* aggiungendogli 1 e cambiando così il comportamento dell’applicazione. Quindi, anche la chiamata a questa funzione va inibita per poter analizzare correttamente l’eseguibile con il debugger.

# Prima patch

Come prima patch effettuata all’eseguibile, si inibisce il controllo effettuato invocando *IsDebuggerPresent* sostituendo la CALL alla API con l’istruzione XOR EAX, EAX. In questo modo, si azzera il registro in cui veniva salvato il valore di ritorno di *IsDebuggerPresent* e la *showWindowIfNoDebugger* effettuerà sempre la *showWindow*, senza mai chiamare *ExitProcess.*

Inoltre, viene negata nella *WndProc* la chiamata alla funzione *checkBeingDebugged* semplicemente sostituendo i bytes dell’istruzione di CALL con delle NOP.

Provando a lanciare l’eseguibile patchato su macchina virtuale (senza utilizzo del Debugger), si nota che la finestra dell’applicazione appare normalmente e sembra funzionante; tuttavia, appare anche una message box che segnala un errore e dopo pochi secondi il processo viene terminato.

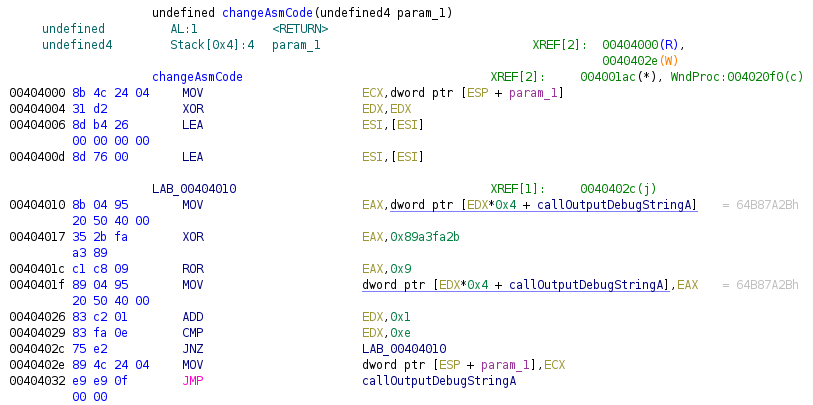


Message box segnalante un errore interno

Inoltre, provando a lanciare l’eseguibile patchato con *OllyDbg*, si nota che la finestra dell’applicazione appare, ma non è disegnata correttamente e il debugger va in crash terminando l’esecuzione. È molto probabile che ciò sia dovuto a qualche altra misura anti-debugger presente; d’altronde, avendo scoperto precedentemente che il programma usa *LoadLibraryA* e *GetProcAddress* per ottenere l’indirizzo di *OutputDebugStringA*, ci si aspetta che tale API venga usata da qualche parte per ostacolare l’analisi tramite debugger.

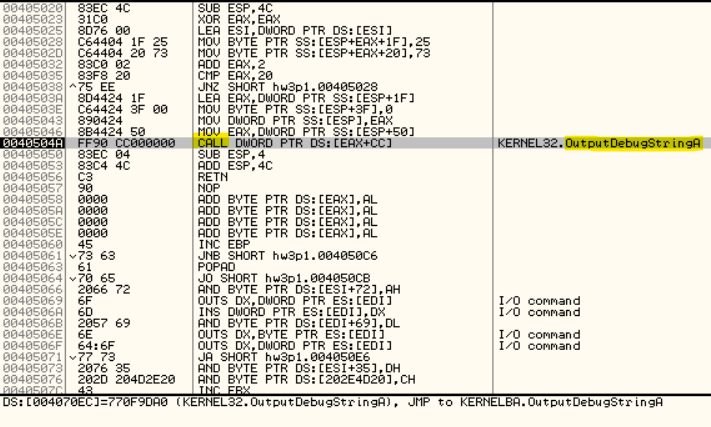
Seguendo l’esecuzione del programma a piccoli passi, impostando opportuni breakpoints, ci si accorge che il debugger va in crash non appena viene invocata la *ShowWindow* all’interno della funzione *showWindowIfNoDebugger*. Da ciò si desume che il problema possa essere presente all’interno della *WndProc* e, in particolare, nella gestione dei messaggi WM\_SIZE e/o WM\_CREATE, ricevuti dalla finestra proprio quando si cerca di mostrarla.

Si è analizzata prima la gestione di WM\_SIZE, in quanto più breve e veloce da analizzare. L’unica funzione sospetta poteva essere la FUN\_00404000, successivamente ridenominata *changeAsmCode*.



Disassemblato della funzione changeAsmCode(): deoffusca delle istruzioni e alla fine le esegue effettuando una jump alla label ridenominata callOutputDebugStringA

Questa funzione effettua un loop in cui, attraverso operazioni di XOR e di ROR, cambia i bytes a partire da una certa label (in seguito ridenominata “callOutputDebugStringA”) in poi. Dopodiché, effettua una jump a tale label. Per scoprire che cosa realmente fa questa funzione, se ne segue il flusso con il debugger e il codice deoffuscato ed eseguito risulta essere il seguente, in cui è evidente una chiamata ad *OutputDebugStringA*:



Chiamata ad OutputDebugStringA("%s%s%s%s%s%s%s%s%s%s")

Il parametro passato alla *OutputDebugStringA* è la stringa “%s%s%s%s%s%s%s%s%s%s”, costruita tramite un loop scrivendo i bytes 0x25 (‘%’) e 0x73(‘s’).

Tale chiamata è quella che fa andare in crash il debugger e va quindi patchata.

# Seconda patch

Per evitare che venga chiamata *OutputDebugStringA*, si decide di sostituire la call a *changeAsmCode* all’interno della gestione di WM\_SIZE nella *WndProc* semplicemente con delle NOP.

Tuttavia, lanciando il programma con la seconda patch senza l’uso del debugger, compare sempre la message box che segnala un internal error e il programma termina dopo pochi secondi. Lanciando l’eseguibile sotto il controllo del debugger, questa volta la finestra per programmare lo shutdown di windows appare disegnata correttamente; tuttavia, anche in questo caso viene mostrata la message box con il messaggio d’errore e, provando ad eseguire senza breakpoint il programma, esso non rimane attivo come ci si aspetterebbe (dovrebbe eseguire il message loop), bensì raggiunge lo stato di terminato.

È evidente che ci sia qualche ulteriore controllo che termini il processo. Inoltre, eseguendo tramite debugger, la terminazione del processo non sembra essere effettuata dal main thread, in quanto si entra nel message loop, ma improvvisamente, dopo un po’ di tempo, il flusso si interrompe e viene terminato il programma.

Dunque si continua ad analizzare la *WndProc* staticamente, concentrandosi sulla gestione del messaggio WM\_CREATE.

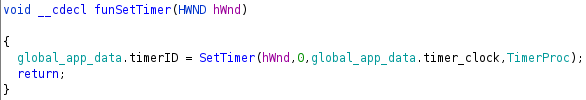


Decompilato della WndProc(); gestione del messaggio WM\_CREATE

Ci sono 3 funzioni degne di nota:

* *CalculateTime()*: è una funzione chiamata per calcolare il tempo dello shutdown e aggiornare i valori delle sottofinestre dei giorni, ore e minuti. Non presenta nulla di strano.
* *funSetTimer()*
* *CalculateFingerprint()*

La *funSetTimer* permette di individuare un ulteriore campo della struct contenente i dati usati dall’applicazione, ovvero il timerID del timer associato alla finestra dell’app. Inoltre, permette di individuare la *TimerProc*.



Decompilato della funzione funSetTimer()

Siccome, per quanto descritto prima, si sospetta che la terminazione del programma sia opera di un thread diverso dal thread main, il thread della *TimerProc* sembra essere un buon candidato.



Decompilato della TimerProc()

Analizzando la *TimerProc*, si individuano due informazioni interessanti: in primis, viene individuato il campo della struct globale in cui è salvato l’indirizzo della funzione da invocare al momento dello scadere del countdown programmato per lo spegnimento di Windows (campo ridenominato “functionToBeCalled”). Tornando indietro alla funzione *init\_data* e alla *WinMain*, se ne individua il valore assegnato e si ridenomina la funzione come “*targetFun*”; tuttavia, il suo disassemblaggio sembra essere stato ostacolato da delle tecniche anti-disassembling e si decide di focalizzarsi su questa funzione in seguito, e di dare priorità all’inibizione delle tecniche anti-debugger per poter così effettuare analisi dinamica avanzata.

La seconda informazione che si desume dall’analisi della *TimerProc* è una chiamata ad una funzione, ovvero la FUN\_004042A0 (successivamente ridenominata *errorWindowAndExitProcess*). Entrando con *Ghidra* nel codice di tale funzione, si rileva che il disassemblaggio è errato a causa di una tecnica anti-disassembler, in particolare la tecnica dell’*always-true condition*: sono presenti dei jump condizionali sempre presi, ma il disassembler, analizzando anche il caso in cui non venissero presi, riconosce delle istruzioni di tipo CALLF che depistano il disassemblaggio. Ripulendo il codice disassemblato, disassemblandolo nuovamente in maniera opportuna e sostituendo i bytes mai eseguiti che portavano ad indentificare le CALLF con delle NOP, si ottiene una disassemblato pulito della funzione e, quindi, anche un decompilato:



Decompilato della funzione errorWindowAndExitProcess(); il campo fingerprint della struct e la variabile globale correct\_fingerprint sono stati individuati in un momento successivo

Si nota che, se la condizione di un if-statement risulta essere vera, si eseguono delle operazioni, tra cui la creazione della message box che riporta l’internal error vista in precedenza e viene invocata la *ExitProcess* che termina il processo.

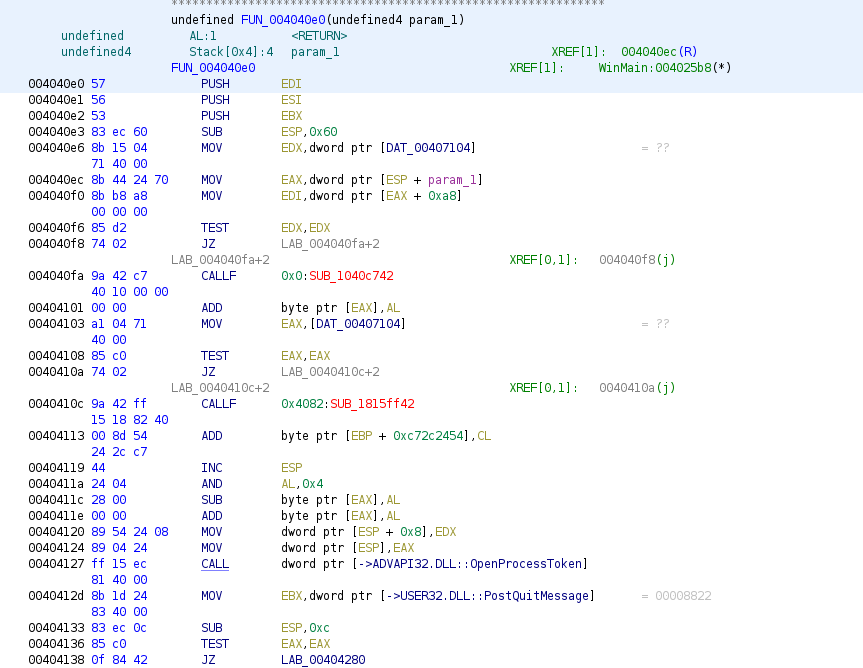
Bisogna effettuare una ulteriore patch che neghi l’invocazione della *errorWindowAndExitProcess* nella *TimerProc*.

# Terza patch

Si sostituiscono i 5 bytes nella *TimerProc* corrispondenti alla CALL a *errorWindowAndExitProcess* con 5 NOP.

Lanciando ora l’eseguibile patchato, non appare più la message box segnalante l’internal error e il programma non termina automaticamente dopo pochi secondi. Anche l’esecuzione del programma controllata dal debugger sembra non presentare problemi: il programma rimane running una volta lanciato e si riesce ad interagire con la finestra dell’applicazione. Si può dunque procedere con l’analisi dinamica avanzata per cercare di scovare il codice di sblocco che rende funzionale il programma.

Dunque, partendo dalla *WinMain,* si segue su *Ghidra* la *targetFun*, ovvero la funzione chiamata quando scade il timeout associato alla finestra dell’applicazione.



Disassemblato della targetFun(): è stata usata la tecnica anti-disassembling dell'always-true condition, infatti DAT\_00407104 è sempre pari a 0

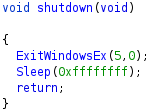
Ripulendo il codice mal disassemblato da *Ghidra* e rieffettuandone il disassembling correttamente in maniera guidata, si riesce ad analizzare anche staticamente la *targetFun*. Si nota che sono presenti chiamate ad API quali GetCurrentProcess, OpenProcessToken, LookupPrivilegeValueA e AdjustTokenPrivileges che sostanzialmente hanno lo scopo di prepararsi ad effettuare lo shutdown del sistema, in particolare si ottiene un access token con i privilegi necessari per effettuare lo spegnimento del sistema.

Successivamente, invocando l’API *GetDlgItemTextA*, viene salvato in un buffer l’input che l’utente ha inserito nella edit box del codice di sblocco necessario per far funzionare correttamente l’applicazione; viene inoltre salvata in una variabile anche la lunghezza di tale input.

Dopodiché, viene eseguito un loop in cui, ancora una volta, vengono effettuate delle operazioni di XOR e ROR di bytes a partire da un indirizzo di una funzione, chiamata FUN\_00405c0. Terminato il ciclo viene invocata tale funzione, passandole 3 parametri: la stringa inserita in input dall’utente, la lunghezza della stringa e l’indirizzo di una label. Seguendo la label, forzando *Ghidra* a disassemblare, si scopre che si tratta di una funzione in cui, ancora una volta, sono state inserite delle misure anti-disassemblaggio con la tecnica dell’*always-true condition*. Ripulendo il codice, si scopre che tale funzione non fa altro che effettuare lo shutdown del sistema operativo invocando l’API *ExitWindowsEx* con parametri 5 (EWX\_SHUTDOWN | EWX\_FORCE) e 0 (nessuna motivazione indicata per lo shutdown). Dunque, tale funzione viene ridenominata *shutdown*.



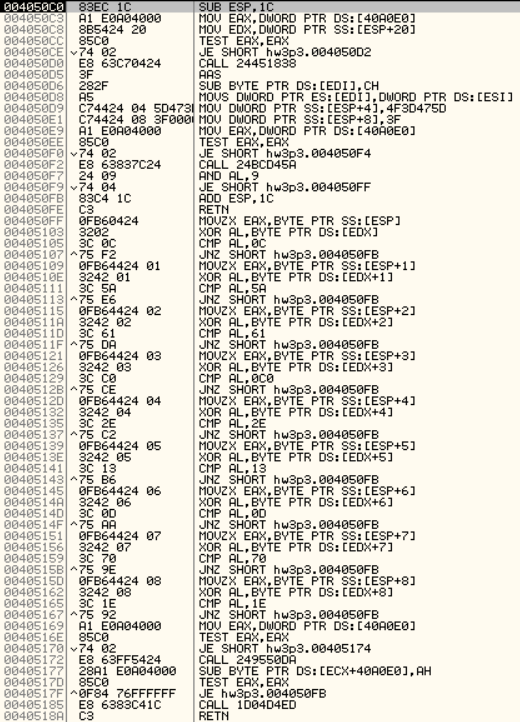
Decompilato della funzione targetFun(); nella figura è evidenziato il punto in cui viene effettuato il deoffuscamento del codice della funzione FUN\_004050c0 e la sua successiva invocazione



Decompilato della funzione shutdown()

# Individuazione del codice di sblocco

Dunque, per scoprire il codice di sblocco, non rimane che seguire con *OllyDbg* il codice della FUN\_004050c0 una volta che è stato ricostruito correttamente. Si setta un breakpoint nella *TimerProc*, al punto in cui viene invocata la *targetFun*, per seguirne l’esecuzione. Finita la procedura di ricostruzione del codice della FUN\_004050c0, quando tale funzione viene invocata, si fa uno *step into* per riuscire a vederne il codice:

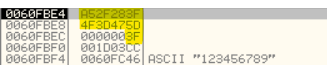


Codice della FUN\_004050c0 visto da OllyDbg dopo essere stato deoffuscato dall'applicazione

In una prima iterazione, si nota che il codice controlla se il valore corrispondente alla lunghezza dell’input sia pari a 9 e, nel caso in cui così non fosse, la funzione ritorna al chiamante senza effettuare lo shutdown.

Dunque, in una seconda iterazione, si prova ad inserire come codice di sblocco la stringa “123456789”, lunga 9 caratteri, e si continua ad analizzare l’esecuzione tramite debugger. Si nota che viene caricato volta per volta in EAX un singolo byte presente sullo stack e se ne fa lo XOR con il corrispondente carattere dell’input dell’utente. Il risultato dello XOR viene confrontato con uno specifico byte che, ad esempio, nel caso del primo carattere deve essere pari a 0C (in esadecimale). Quindi, il confronto sulla correttezza del codice di sblocco non viene effettuato “in chiaro”, ma viene “cifrato” l’input dell’utente effettuando lo XOR con una “chiave” di 9 bytes e viene controllata la correttezza di tale “ciphertext”.

Guardando sullo stack, si riesce ad individuare facilmente la “chiave” che viene usata:



La "chiave" caricata sullo stack: 3F 28 2F A5 5D 47 3D 4F 3F

È analogamente semplice riuscire ad individuare il “ciphertext” atteso, guardando alle singole istruzioni di CMP presenti nella funzione FUN\_004050c0. Dunque, si ha:

* chiave = 3F 28 2F A5 5D 47 3D 4F 3F
* ciphertext = 0C 5A 61 C0 2E 13 0D 70 1E
* plaintext ⊕ chiave = ciphertext 🡪 ciphertext ⊕ chiave = plaintext

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3F | 28 | 2F | A5 | 5D | 47 | 3D | 4F | 3F |

⊕

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0C | 5A | 61 | C0 | 2E | 13 | 0D | 70 | 1E |

=

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 33 | 72 | 4E | 65 | 73 | 54 | 30 | 3F | 21 |

↓

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ‘3’ | ‘r’ | ‘N’ | ‘e’ | ‘s’ | ‘T’ | ‘0’ | ‘?’ | ‘!’ |

Consultando la tabella di codifica ASCII, si è convertito il plaintext ottenuto al corrispettivo valore in caratteri e il codice di sblocco risulta essere la stringa “3rNesT0?!”.

# Ulteriori analisi sul programma

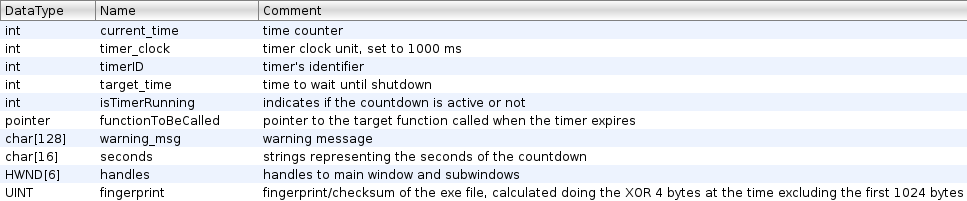
Nella parte della *WndProc* in cui viene gestito il messaggio WM\_CREATE, era stata precedentemente individuata una funzione chiamata subito dopo la *funSetTimer*: si decide di approfondire l’analisi di tale funzione che, successivamente, è stata ridenominata *CalculateFingerprint*.



Decompilato della funzione CalculateFingerprint()

Questa funzione utilizza i valori *startingAddress* e *nSizeFileLow* che erano stati precedentemente calcolati tramite l’invocazione della funzione *getFileInfo* analizzata precedentemente. Sostanzialmente, ciò che la funzione fa è sommare allo *startingAddress* il valore 1024 e iterare 4 bytes alla volta, fino alla fine del programma, calcolando ad ogni iterazione lo XOR dei 4 bytes considerati in quell’iterazione con il risultato ottenuto all’iterazione precedente. Sostanzialmente, in questo modo, non ottiene nient’altro che un fingerprint del file eseguibile. Tale fingerprint, viene salvato in uno dei campi della struct globale contenente le informazioni principali usate dall’applicazione.

Dunque, adesso la struct è stata analizzata in tutti i suoi campi ed è riassunta dalla figura riportata di seguito:

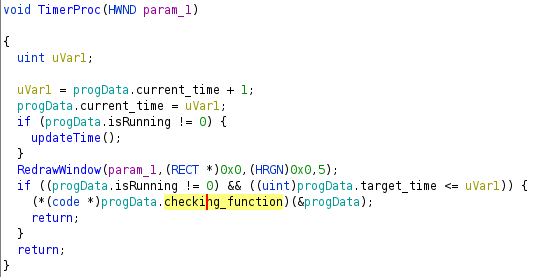


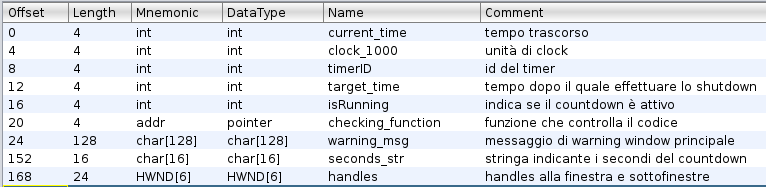
Struct "app\_data" e i campi che la compongono

A questo punto dell’analisi, si è tornati sulla funzione *TimerProc* e, con le informazioni ottenute sulla struttura dati usata dal programma, ci si rende conto che la CALL all’indirizzo nel segmento dati trovata inizialmente è una CALL ad un campo della struct. Dunque, tale campo contiene l’indirizzo di una funzione, alla quale viene passato come parametro un riferimento alla struttura dati globale. Per indagare su quale funzione fosse, si ritorna indietro alla *WinMain*, che passa come parametro alla *fn\_init\_data* una variabile globale. Tale variabile viene attribuita al campo della struct di cui si effettua la call nella *TimerProc*.

A questo punto, la struttura dati *program\_data* usata dall’applicazione è ricostruita. Con ogni probabilità, la funzione che viene invocata è quella che effettua il controllo sulla correttezza del codice di sblocco ed esegue lo shutdown di Windows.

Si riportano di seguito il decompilato della *TimerProc* e la struttura dati *program\_data* definita in *Ghidra,* dopo averne denominato il campo che punta alla funzione come “checking\_function”:

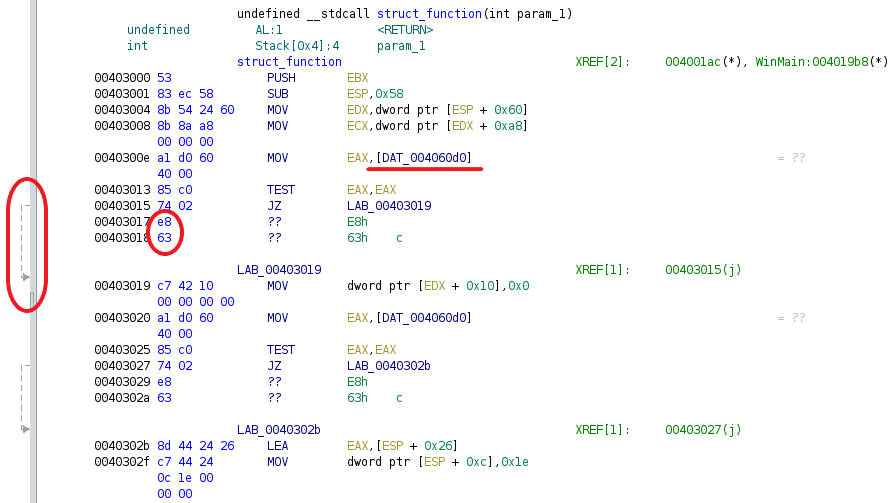




Ora, non resta altro che indagare sulla funzione che viene invocata dalla *TimerProc* e di cui la struct conserva l’indirizzo. Si nota che tale funzione è presente anch’essa nel segmento dati. *Ghidra* non è riuscito a riconoscerla come funzione e quindi i bytes delle istruzioni che la compongono non sono stati disassemblati. Si forza *Ghidra* a disassemblare e a riconoscere la funzione; il risultato ottenuto è il seguente (solo la parte iniziale della funzione è mostrata di seguito):



Come si può notare, vengono segnalate delle istruzioni CALL in rosso, ad indicare che gli indirizzi specificati non sono presenti all’interno del file eseguibile. Analizzando le informazioni fornite dal disassembler di Ghidra, si nota che prima di ognuna di queste CALL, c’è un’istruzione JZ ad una label al terzo byte dei 5 che compongono l’istruzione di CALL successiva (1 byte per l’opcode + 4 bytes per l’indirizzo a 32 bit a cui saltare). Si desume che, molto probabilmente, il disassemblatore vada in confusione e quindi si decide di disassemblare il codice poco alla volta, per blocchi, seguendo le indicazioni fornite dalle labels. Il risultato ottenuto è il seguente:



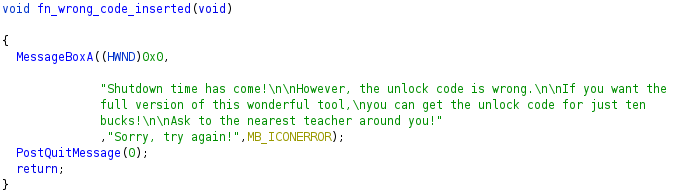
Nell’immagine, è evidenziata una coppia di bytes [E8, 63] che si ripetono di frequente nella funzione, ma che non rappresentano nessuna istruzione; bensì confondevano il disassemblatore di *Ghidra*, inducendolo ad interpretarli come facenti parte di una istruzione di CALL, in quanto l’opcode di tale istruzione è appunto il byte E8. Si nota inoltre che il flusso d’esecuzione giunge a tali bytes solo se il valore della variabile globale DAT\_004060d0 è diverso da zero. Tuttavia, tale variabile è situata nel segmento BSS, dunque è inizializzata a 0; inoltre, non viene mai riferita in scrittura e il suo valore non verrà mai modificato, risultando sempre pari a zero. In conclusione, il salto condizionale corrispondente all’istruzione JZ prima dei due bytes [E8, 63] verrà sempre preso e il flusso di esecuzione non si ritroverà mai ad eseguire tali bytes.

Si suppone che tale struttura sia stata ottenuta utilizzando un offuscatore di codice per ostacolare il reversing del programma.

Si decide di effettuare per ogni occorrenza della coppia di bytes [E8, 63] una Patch Instruction all’interno di *Ghidra*, inserendo una coppia di NOP [90, 90]. A questo punto, la funzione, ridenominata *struct\_method,* diventa molto più leggibile sfruttando il decompilatore. Nell’immagine che si riporta di seguito, figura già una chiamata alla API AdjustTokenPrivileges. In realtà, essa non era stata riconosciuta da *Ghidra* ed è stato necessario effettuare una ulteriore forzatura a disassemblare dopo averne pulito il codice erroneamente disassemblato in precedenza. A questo punto, ne è venuta fuori una semplice istruzione di jump alla API AdjustTokenPrivileges della DLL ADVAPI32 ed è quindi stata indicata come una thunk function. Allo stesso modo, è stato necessario disassemblare la funzione indicata come *fn\_wrong\_code\_inserted*, che fa apparire la finestra contenente il messaggio d’errore quando il codice di sblocco inserito dall’utente risulta essere sbagliato.



Dunque, la *struct\_method* chiama la GetDlgItemTextA per ottenere l’input inserito dall’utente nella edit box del codice di sblocco (5° HANDLE nell’array *handles* della struttura *program\_data*). Dopodiché, controlla che la lunghezza di tale input sia pari a 9 e, char per char, effettua il confronto con il codice di sblocco corretto, il quale risulta essere la stringa “3RnESt0!?”. Se il controllo non va a buon fine, viene invocata la *fn\_wrong\_code\_inserted* per mostrare il messaggio d’errore:



Invece, qualora il codice di sblocco inserito fosse corretto, la funzione si prepara ad effettuare lo shutdown del sistema nel seguente modo:

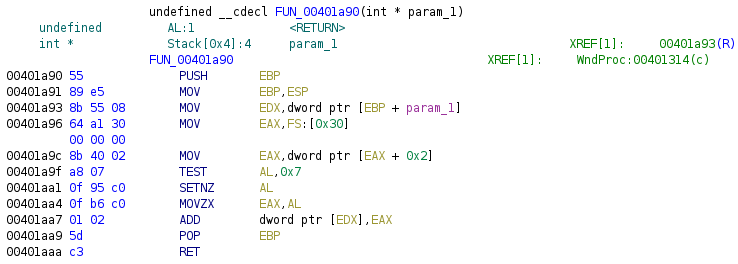
1. Chiama l’API GetCurrentProcess per ottenere uno pseudo handle per il processo corrente;
2. Invoca la OpenProcessToken per aprire un access token associato con il processo corrente, specificando una access mask pari a 0x28, ovvero l’or delle macro TOKEN\_ADJUST\_PRIVILEGES (0x20) e TOKEN\_QUERY (0x08). Non essendo presenti sulla documentazione Microsoft, tali valori sono stati reperiti al seguente link: <https://www.pinvoke.net/default.aspx/advapi32.openprocesstoken>. Tali macro indicano che è possibile effettuare delle query sull’access token e che se ne possono abilitare o disabilitare i privilegi;
3. Effettua una chiamata all’API LookupPrivilegeValueA con la quale recupera il Locally Unique Identifier (LUID) del privilegio SE\_SHUTDOWN\_NAME, identificato dalla stringa “SeShutdownPrivilege”, e lo salva nel campo Privileges di una struct locale di tipo TOKEN\_PRIVILEGES. In questo modo, ha ottenuto l’identificatore sulla macchina locale del privilegio per effettuare lo shutdown del sistema;
4. Il campo Privileges della struct TOKEN\_PRIVILEGES è a sua volta una struct, di tipo LUID\_AND\_ATTRIBUTES, che ha come primo campo il LUID del privilegio e come secondo campo un attributo, che viene settato a 2, ovvero a SE\_PRIVILEGE\_ENABLED;
5. L’indirizzo della struct TOKEN\_PRIVILEGES viene passato, insieme all’handle al token ottenuto con la OpenProcessToken, alla API AdjustTokenPrivileges per abilitare il privilegio necessario ad effettuare lo shutdown per l’access token appena aperto;
6. Se tutto è andato a buon fine, viene effettuato lo spegnimento del sistema invocando l’API ExitWindowsEx, passando come primo parametro 5 e come secondo parametro 0. Il primo parametro corrisponde all’or dei due flag EWX\_SHUTDOWN (0x01) ed EWX\_FORCE (0x04): ciò vuol dire che il sistema viene arrestato forzando tutte le applicazioni a terminare, comportando una potenziale perdita di dati da parte delle applicazioni. Il secondo parametro, pari a 0, indica che non viene indicato alcun motivo per l’inizio dello shutdown.

# Contromisure anti-debugging

Come ultima operazione nel processo di analisi e reversing dell’eseguibile, si ritorna su una funzione chiamata all’interno della *WinProc* inizialmente ignorata:



A questa funzione FUN\_00401a90, viene passato come parametro di ingresso l’indirizzo di uMsg, ovvero l’argomento della funzione *WinProc* che rappresenta il codice identificativo del messaggio ricevuto dalla finestra. Di seguito viene riportato il disassemblato della funzione:



In sistemi con architettura x86, il registro FS è usato per accedere tramite offset ad una struttura dati detta TIB (Thread Information Block), o anche TEB (Thread Envirnment Block), contenente informazioni sul thread attualmente in esecuzione. In particolare, con FS:[0x30], ovvero con offset 0x30 a partire da FS, si ottiene l’indirizzo di una ulteriore struttura dati detta PEB (Process Environment Block), come suggerito dalla pagina Wikipedia: <https://en.wikipedia.org/wiki/Win32_Thread_Information_Block>.

La struttura dati PEB contiene informazioni sul processo corrente, ma non è completamente documentata da Windows. Tuttavia, la funzione tenta di recuperare il valore ad offset 2 di tale struttura, che, come desunto dalle risorse web <https://www.aldeid.com/wiki/PEB-Process-Environment-Block/BeingDebugged> e <https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb>, corrisponde al campo BeingDebugged. Tale campo è un byte che è pari a 1 nel caso in cui il processo corrente è eseguito in modalità di debug, 0 altrimenti.

Dunque, ciò che la funzione chiamata dalla *WndProc* fa, è di consultare tale valore e, nel caso in cui l’applicazione sia stata lanciata con un debugger, modifica il valore di uMsg aumentandolo di 1 prima che ne venga fatto il confronto con le varie macro corrispondenti ai tipi di messaggi che la finestra deve gestire. Altrimenti, il valore di uMsg resta inalterato. In questo modo, si ostacola l’analisi dinamica avanzata del programma, facendogli assumere un comportamento completamente diverso quando eseguito tramite un debugger.

# Conclusione

In conclusione, il codice di sblocco del programma è “3RnESt0!?”. Esso è stato anche testato su macchina virtuale ed ha riportato il risultato atteso, ovvero lo spegnimento del sistema. Il sistema viene spento forzando tutte le applicazioni attive a terminare, causando una possibile perdita di dati delle applicazioni, e senza specificare la ragione dello shutdown.

L’applicazione usa una struttura dati un po’ “anomala”, in quanto è una struct che ha un campo contenente l’indirizzo di una funzione, facendo pensare ad una classe in un contesto Object Oriented. Inoltre, tale funzione è contenuta nella sezione dati dell’eseguibile e probabilmente è stata modificata utilizzando delle tecniche di offuscamento per complicarne il reversing.

Infine, si riporta che l’applicazione tenta di proteggersi da un’analisi effettuata con un debugger, rilevando tale situazione e mutando il suo comportamento a runtime, così da ingannare l’analista.